

IB015 Neimperativní programování

Jiří Barnat

Organizace kurzu

Kurz IB015

- Zakončen zkouškou.
- 6 kreditů ($2/1/1+zk$) = 180 hodin = 22 pracovních dnů

Přednáška

- Fyzicky prezenčně dle rozvrhu.
- Výuková videa v interaktivní osnově.

Cvičení

- Se 14 denní periodou v určené časové sloty.
- Týdně alternují suché a liché skupiny.
- Výjimka – první týden, cvičení mají všichni.

Samostatné domácí úlohy

- Zadávány skrze interaktivní osnovu předmětu.

Závěrečná písemná zkouška

- Povinná část – Test Minimálních Dovedností (TMD)
- Nepovinná část, možno získat 10 bodů.

Domácí úlohy

- Každý týden, vzhledem ke 14 dennímu cyklu cvičení alternují
 - úkol před cvičením (0,5 bodu).
 - úkol po cvičení (1–1,5 bodu).
- Celkem možno získat 15 bodů.

Forma DÚ

- Odpovědník v ISu
 - časové omezení na vypracování.
 - **možné otevřít pouze jednou, otvírejte po přednášce!**
- Programovací úloha (3x, v druhé půlce semestru).

Neúspěšné ukončení

- Nezískání alespoň 8 bodů z DÚ – hodnocení X.
- Nesplnění TMD – hodnocení F.
- Splnění TMD, nezískání 10 a více bodů – hodnocení F.
- Opravné termíny na TMD dle SZŘ.

Úspěšné ukončení

- Zisk alespoň 8 bodů z DÚ.
- Splnění TMD.
- Znamka dle počtu bodů za DÚ a zkoušku:
E:[10,12) D:[12,14) C:[14,17) B:[17,20) A:[20,25]
- 12+ bodů z DÚ a úspěšné ukončení = odmazání jednoho F.

Cíle kurzu

- Studenti se seznámí s funkcionálním a logickým paradigmatem programování, díky čemuž se odprostí od imperativního způsobu uvažování o problémech a jejich řešení.
- V rámci kurzu se studenti blíže seznámí s funkcionálním programovacím jazykem Haskell a s logickým programovacím systémem Prolog.

Schopnosti absolventa

- Je schopen dekomponovat výpočetní problém na jednotlivé funkce a tuto schopnost používá při vytváření vlastních kódů i v imperativních programovacích jazycích.
- Umí efektivně použít prvky funkcionálního programování v imperativních programovacích jazycích.
- Má základní znalost programovacích jazyků Haskell a Prolog.
- Rozumí způsobu popisu programů ve funkcionálním a logickém výpočetním paradigmatu.
- Umí oddělit CO od JAK.

Předpoklady

- Možné úspěšně absolvovat bez znalosti programování.
- Schopnost abstraktního myšlení.
- Základní počítačová gramotnost (Unix/Linux OS).

Znalost imperativního programování

- Je výhodou pro pochopení rozdílného způsobu myšlení v imperativním a neimperativním světě.
- Může být zpočátku mentální bariérou.

Funkcionální paradigma

- <http://haskell.cz/>
- Thompson, Simon. Haskell: the craft of functional programming.
- Structure and Interpretation of Computer Programs
[<http://mitpress.mit.edu/sicp/full-text/book/book.html>]

Logické paradigma

- <http://www.learnprolognow.org>
- Nerode, Shore: Logic for Applications

Co znamená programovat?

Programování

- Vytvoření a zápis postupu řešení problému s takovou úrovní detailů a přesnosti, aby tento popis mohl být mechanicky vykonáván strojem, zejména počítačem.
- Zápis postupu = zdrojový kód programu.
- Zdrojový kód programu je uložen v textovém souboru.

Programovací jazyk

- Uměle vytvořený jazyk pro přesný a jednoznačný zápis programů člověkem.

Schopnost programovat

- **Mentální schopnost nacházet mechanicky proveditelné postupy za účelem řešení daného problému.**
- Schopnost přesně formulovat postupy v daném programovacím jazyce.

Volba a znalost programovacího jazyka

- Programovacích jazyků je mnoho.
- Volba programovacího jazyka klade omezení na způsob formulace zamýšlených postupů.

Riziko a klam moderní doby

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné poznání syntaxe a sémantiky programovacího jazyka nedělá dokonalého programátora.

Riziko a klam moderní doby

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné poznání syntaxe a sémantiky programovacího jazyka nedělá dokonalého programátora.

Nedokonalé vs. dokonalé

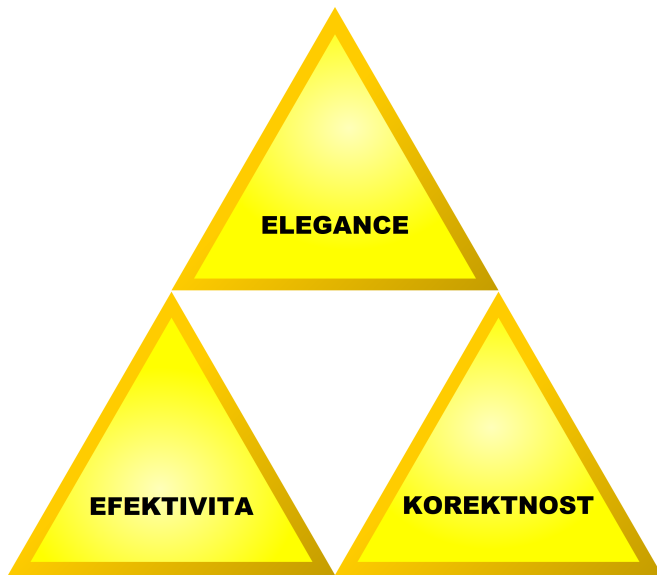


Riziko a klam moderní doby

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné poznání syntaxe a sémantiky programovacího jazyka nedělá dokonalého programátora.

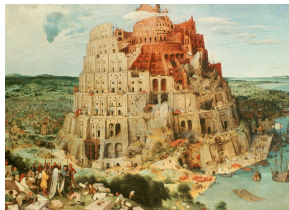
Nedokonalé vs. dokonalé





Klasifikace

- Imperativní — C/C++, Java, Python, ...
- Funkcionální – **Haskell**, OCaml, ...
- Logické – **Prolog**, ...



Jakým jazykem mluví počítač?

- Strojový kód. Program ve strojovém kódu je posloupnost čísel.
- Pro spuštění programu je potřeba provést překlad zdrojového kódu programu do strojového kódu procesoru.
- Překlad se realizuje pomocí **překladače** nebo **interpretu**.
- Pro každý programovací jazyk je potřeba jiný překladač/interpret.

Překladač

- Pro soubor se zdrojovým kódem programu vytvoří soubor obsahující popis programu ve strojovém kódu.
- Výsledný soubor je spustitelný.
- Pracuje se soubory.

Interpret

- Pro daný výraz / příkaz vytvoří odpovídající překlad do strojového kódu a ihned jej provede.
- Nevytváří výsledný spustitelný soubor.
- Často má možnost pracovat interaktivně.
- Pracuje s jednotlivými příkazy/výrazy.

Programovací jazyk Haskell

- Překladač – `ghc`.
- Interaktivní interpret – `ghci`.

Překladače programovacího jazyka C/C++

- GNU C++ Compiler (`g++`, `gcc`)
- Intel C++ Compiler
- Microsoft Visual C++ Compiler

Programujeme pomocí funkcí

Funkce v programování

- Funkce je předpis jak z nějakého vstupu vytvořit výstup.
- Transformace vstupů na výstupy musí být jednoznačná.

Příklady funkcí

- $f(x) = x * (x + 2)$
- `objemKvadr(a, b, c) = a * b * c`
- ...

Typ funkce

- Vymezení objektů, se kterými daná funkce pracuje a které vrátí na výstup, je součástí definice funkce. Mluvíme o tzv. **typu funkce**.


Příklady

- Funkce, která otočí obrázek o 90 stupňů směrem vpravo.
`rotuj90vpravo :: Obrázek -> Obrázek`
- Objem kvádru.
`objemKvadrů :: Číslo × Číslo × Číslo -> Číslo`
- Počet hran polygonu.
`hranyPolygonu :: Polygon -> CeléČíslo`



Předpoklady


rotuj90vpravo :: Obrázek -> Obrázek

hranyPolygonu :: Obrázek -> CeléČíslo

 :: Obrázek

Aplikace funkcí

rotuj90vpravo  \rightsquigarrow 

hranyPolygonu  \rightsquigarrow 3

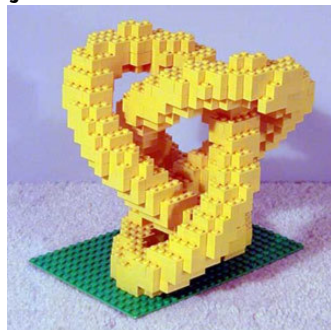
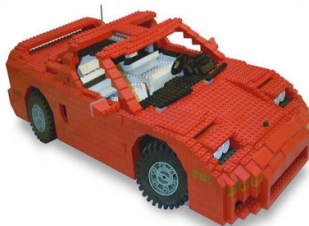
Pozorování

- Složitější úkony lze realizovat pomocí jednodušších operací.
- Složitější funkce lze definovat **složením** jednodušších.

Pozorování

- Složitější úkony lze realizovat pomocí jednodušších operací.
- Složitější funkce lze definovat **složením** jednodušších.

Skládání – cesta ke složitějším objektům a funkcím





Operátor .

- $(f1 . f2) x = f1 (f2 x)$
- Čteme jako „f1 po f2“.

Příklad

- Předpokládejme funkci `zdvoj`, která vezme obrázek a vytvoří nový zkopírováním vloženého obrázku dvakrát vedle sebe.

`zdvoj :: Obrázek -> Obrázek`

`zdvoj`  \rightsquigarrow 

- Novou funkci `rotujAZdvoj` můžeme definovat takto:

`rotujAZdvoj :: Obrázek -> Obrázek`



`rotujAZdvoj x = (zdvoj . rotuj90vpravo) x`


`rotujAZdvoj`  \rightsquigarrow 


(rotujAZdvoj . rotujAZdvoj) \triangle \rightsquigarrow

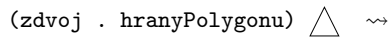
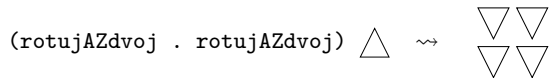
((zdvoj . zdvoj) . zdvoj) \triangle \rightsquigarrow



(zdvoj . hranyPolygonu) \triangle \rightsquigarrow

`(rotujAZdvoj . rotujAZdvoj)`  \rightsquigarrow 


`((zdvoj . zdvoj) . zdvoj)`  \rightsquigarrow


`(zdvoj . hranyPolygonu)`  \rightsquigarrow

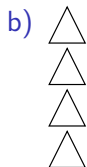
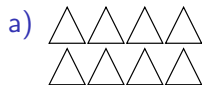



`(rotujAZdvoj . rotujAZdvoj)`  \rightsquigarrow 

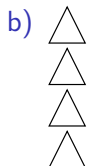
`((zdvoj . zdvoj) . zdvoj)`  \rightsquigarrow 

`(zdvoj . hranyPolygonu)`  \rightsquigarrow **ERROR**

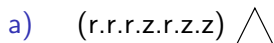
Jak pomocí zdvoj, rotuj90vpravo a  vyrobit následující?



Jak pomocí zdvojení, rotace 90° vpravo a  vyrobit následující?



Řešení



Složené funkce a η -redukce

- Složení funkcí je možné definovat bez uvedení parametru.
- Tj. definici

`rotujAZdvoj x = (zdvoj.rotuj90vpravo) x`

lze zapsat také jako

`rotujAZdvoj = zdvoj.rotuj90vpravo`

POZOR na prioritu vyhodnocování v Haskellu

- Aplikace funkce na parametry má nejvyšší prioritu.

`zdvoj.rotuj90vpravo \triangle = zdvoj.(rotuj90vpravo \triangle) \rightsquigarrow ERROR`

- Závorky kolem výrazu `zdvoj.rotuj90vpravo` jsou při aplikaci na hodnotu \triangle nutné.

Deklarace typu funkce (typová signatura):

rotujAZdvoj :: Obrázek ->Obrázek

Jméno funkce

rotujAZdvoj x = (zdvoj.rotuj90vpravo) x

Tělo funkce

rotujAZdvoj x = (zdvoj.rotuj90vpravo) x

Definice funkce

rotujAZdvoj x = (zdvoj.rotuj90vpravo) x

Formální parametr

rotujAZdvoj x = (zdvoj.rotuj90vpravo) x

Aktuální parametr

rotujAZdvoj △

Výraz

rotujAZdvoj △

Podvýraz

rotujAZdvoj (rotujAZdvoj △)

Funkcionální programování v Haskellu

Funkcionální výpočetní paradigma

- program = výraz + definice funkcí
- výpočet = úprava (zjednodušení) výrazu
- výsledek = hodnota (nezjednodušitelný tvar výrazu)

Příklad programu

- definice funkcí

```
square x = x * x
```

```
pyth a b = square a + square b
```

- výraz

```
pyth 3 4
```

Program

- definice funkcí

```
square x = x * x
```

```
pyth a b = square a + square b
```

- výraz

```
pyth 3 4
```

Výpočet

pyth 3 4 \rightsquigarrow square 3 + square 4 \rightsquigarrow 3 * 3 + square 4 \rightsquigarrow

\rightsquigarrow 3 * 3 + 4 * 4 \rightsquigarrow 9 + 4 * 4 \rightsquigarrow 9 + 16 \rightsquigarrow

\rightsquigarrow 25

Lokální definice

- Definují symboly (funkce, konstanty) pro použití v jednom výrazu, vně tohoto výrazu jsou tyto symboly nedefinované.
- Lokální definice mají vyšší prioritu než globální definice.
- V jazyce Haskell možno dvěma způsoby:
 - let lokální definice in výraz
 - definice where lokální definice

Příklad lokální definice pomocí let ... in

- `let fcube x = x * x * x in fcube 12`
`let fcube x = x * x * x in let c = 12 in fcube c`
`let fcube x = x * x * x; c = 12 in fcube c`
- Konstrukce let ... in vytváří nový výraz.

Příklad lokální definice pomocí `where`

- Konstrukce `where` musí být součástí definice, nevytváří výraz.
- **Úkol:** Zjisti obvod pravoúhlého trojúhelníku z délek odvěsen.
- `obvodZOdvesen :: Float -> Float -> Float`

```
obvodZOdvesen a b = a + b + c
```

```
  where square x = x * x
```

```
        c = sqrt (square a + square b)
```

Pravidla pro zarovnání

- Dlouhé řádky se zalomí a pokračují s odsazením.
- Definice v jednom rozsahu musí začínat na stejném sloupku.
- Odsazení mohou libovolně zvětšovat, nemohou zmenšovat, chci docílit maximální čitelnost a srozumitelnost kódu.

Čísla

- `Integer` – libovolně velká celá čísla
- `Int` – celá čísla do velikosti slova procesoru
- `Float` – reálná čísla
- `Rational` – racionální čísla

Znaky a řetězce

- `Char` – znak, příklady hodnot: `'a'`, `'2'`, `'>'`
- `String` – řetězec, například: `"Toto je řetězec."`
- `String` je totéž co `[Char]`

Pravdivostní hodnoty

- `Bool`
- Typ `Bool` má pouze 2 hodnoty: `True` a `False`

Příklad

- Definujte funkci `jedna_nebo_dva`, která vrátí `True` pokud dostane na vstupu číslo 1 nebo 2, jinak vrátí `False`.

```
jedna_nebo_dva :: Integer -> Bool
jedna_nebo_dva 1 = True
jedna_nebo_dva 2 = True
jedna_nebo_dva _ = False
```

Definice funkcí s více definičními rovnostmi

- Na místě formálních parametrů se použijí tzv. vzory.
- Použije se první vzor, který vyhovuje, nic jiného.
- Symbol `_` vyhovuje libovolnému parametru.
- Lze použít pro větvení výpočtu.

Podmíněný výraz

- if *podmínka* then *výraz1* else *výraz2*
- *podmínka* – výraz, který se vyhodnotí na hodnotu typu `Bool`
- *výraz1* se vyhodnotí pokud se podmínka vyhodnotí na hodnotu `True`, *výraz2* se vyhodnotí, pokud se podmínka vyhodnotí na hodnotu `False`.
- Výrazy *výraz1* a *výraz2* musejí být stejného typu.

Test na rovnost

- Pro dotaz na rovnost používáme symbol `==`.
- `3 == 4` \rightsquigarrow `False`
- `3 = 4` \rightsquigarrow **Error**

Možnosti zápisu binárních funkcí

- Prefixový zápis binárních funkcí: $(+) 3 4$, $\text{min } 3 5$
- Infixový zápis binárních funkcí: $3+4$, $3 \text{ 'min' } 5$

Volání funkce a parametry

- Jméno funkce a použité parametry jsou odděleny mezerou, pokud je některý z parametrů výraz, který je sám o sobě aplikace funkce na argumenty, je třeba celý tento výraz ozávkovat.
- $(*) 3 4 + 5 \rightsquigarrow 17$
- $(*) 3 + 4 5 \rightsquigarrow \text{Error}$
- $(*) 3 (+) 4 5 \rightsquigarrow \text{Error}$
- $(*) 3 ((+) 4 5) \rightsquigarrow 27$

Zde byla ukázka jednoduchého programu v Haskellu, který bylo možné přeložit do spustitelného programu a spustit.

Na základě zkušeností a četných žádostí cvičících byl tento příklad z první přednášky odstraněn.

Co to je?

- Úkol, který je možné vyřešit na základě faktů doposud uvedených na přednáškách.
- Slouží ke kontrole, že zvládám to, co bych už měl(a) umět.

Checkpoint

- V programovacím jazyce Haskell napište funkci, která bude řešit dělitelnost dvou celočíselných čísel, tj. pro své dva celočíselné argumenty dělenec a dělitele, rozhodne, zda je zadaný dělenec dělitelný beze zbytku zadaným dělitelem a to tak, že nepoužije operace pro dělení `/` ani počítání zbytku po dělení `mod`, je povoleno použít operaci celočíselného dělení `div`.
- Funkci otestujte s použitím interpretu jazyka Haskell.